IR Project - Travelsearch

March 31, 2019

Jesse Haenen (10670742)
University of Amsterdam

Leon van Veldhuijzen (10817263)
University of Amsterdam

Stijn Uijen (10732969)
University of Amsterdam

ABSTRACT

This paper presents the data acquisition, data processing and storage and basic search functionalities of an information retrieval system implementation called Travelsearch, which eventually will retrieve and rank relevant travel documents from a large textual corpus.

1 INTRODUCTION

The goal of this project is to implement a specialized search system which will retrieve and rank relevant travel articles from a corpus of scraped internet documents. Travelsearch will be a system where a user can enter query, e.g. 'Cambodia' or 'scuba diving', and which then returns relevant information for that destination, activity, or other topics relating to travel. From a selection of four travel related websites, at least ten thousand pages were scraped, processed and indexed. With a basic search engine interface developed with Flask, queries can be entered and relevant documents will be retrieved by means of term-based ranking and semantic search. Several additional functionalities are implemented such as filters by topic and location. The code for this project can be found on an public GitHub repository[1].

2 DATA ACQUISITION

The first phase of building a search system is acquiring the data. This can be done by crawling the web. Crawling is an technique to automatically extract information from internet addresses. As the topic of interest is travel, websites which contain a lot of pages describing travel destinations can be used to gather a corpus of pages. For this research, scraping has been implemented with the help of the Scrapy Python library [2]. Scrapy is an open source framework written in Python for extracting data from websites. Initializing a Scrapy project is straightforward, and the architecture allows adjustments of almost any part of the scraping pipeline and the implementation of self-made 'spiders', which are used to crawl web pages. The initialization of an project automatically creates an repository where all the necessary Python files reside. The following section will describe the functionalities concerning scraping that have been implemented thus far.

2.1 Basic crawling

The functionality of basic crawling means that a single-threaded spider crawls without any politeness policies. Single-threaded crawling means that only one instance of a spider is running on a local machine. The term politeness stands for the general concern of keeping in mind some concern towards the owners of the websites that are being crawled. This will be explained in more detail in section 2.2.

This was implemented in Scrapy by building a custom spider class. In Scrapy, a spider class is used to define how a given site, or multiple sites, has to follow links and how to extract data [3]. In this class, we can specify on which URL the spider has to begin crawling, which links it has to follow, and what data has to be extracted from the HTML or the metadata (like the URL of the page).

2.2 Politeness

According to crawling etiquette, some basic rules have to followed: 1) a online scraper has to identify itself, 2) the scraper has to obey the robots exclusion protocol, which can be found in the *robots.txt* section of a website, and 3) the crawler has to keep a low bandwidth usage in a given web site making sure not to overload the host of the website owner. With Scrapy we can implement politeness by altering the settings in the automatically generated *settings.py* file[4]. Here we can set all three rules as described above by simply defining the following settings:

```
1) BOT_NAME
2) ROBOTSTXT_OBEY
3) DOWNLOAD_DELAY
```

The first setting specifies the bot name is used to construct the User-Agent that is used to announce the identity of the crawler. The second setting ensures that the spider obeys to the *robots.txt* file, and the third setting where specifies the amount the spider has to wait before it scrapes the content of a web page, we set this last setting to 1.5 seconds.

2.3 Distributed crawling

Distributed crawling means that multiple crawlers are working in parallel. This has to implemented if the number of pages that have to be visited becomes large, or the process of crawling should be increased in terms of speed. Problems during parallel crawling can arise when the URLs between threads have to be synchronized, while at the same time minimize communication overhead. Both these problems can be solved by initializing crawlers on different machines, where each crawler remains on a given domain. This ensures that the crawlers do not have

[1]https://github.com/jessefh/travelsearch
[2]https://scrapy.org/
[3]https://docs.scrapy.org/en/latest/topics/spiders.html
[4]https://docs.scrapy.org/en/latest/topics/settings.html

to synchronize URLs at run time, since each individual crawler stays on its given domain(s).

For each of the three websites of interest, a individual spider class was created, where the starting URL and the allowed domain was specified. Each of these spiders were run on different CPU's in parallel.

2.4 Crawling travel websites

To create a substantial corpus, 5 sites are crawled with each 10.150 content pages, i.e. pages containing meaningful information. This resulted in a corpus of in total 50.750 pages. As stated in section these websites are Wikipedia (sovereign countries page), the Lonely Planet website (South America forum), RickSteves (Europe forum), Fodors (Asia forum) and Travellerspoint (general travel forum). The specific (sub)domains where the crawling started, are the following:

- https://www.lonelyplanet.com/thorntree/forums/am...
- https://www.travellerspoint.com/forum
- https://en.wikipedia.org/wiki/List_of_sovereign_states
- https://community.ricksteves.com/travel-forum/all-topics
- https://www.fodors.com/community/asia/

From each of these starting point directories, 10.500 content pages were gathered. Each HTML file was written to disk, and a text file was created with all the URLs. The text file containing all URLs is stored in the folder 'data' and is named *url_list_large.txt*. In order to achieve this, the links the individual spider had to follow had to be tuned for specific websites. For example, on the Wikipedia page with the all the countries listed, the crawler looks at all links on the page and determines (based on explicit rules) whether it should follow that link, then in the second layer, it follows all links. This is done in a breath-first manner and is added to the crawler setting, as we want to scrape general information about every country first, and more in-depth subtopics later. This is especially important given the way Wikipedia is organized; a very broad spectrum of information can be found when following every link on the page. When all countries are stored, the crawler follows all links for every country in the starting page. For the other spiders we incrementally updated the URLs in order to scrape each resulting page.

From testing the results of our basic term-based search described in section 4.2, the website choice of Wikipedia often resulted in results that were not relevant to the query, or generally Wikipedia results being ranked low. This was partially resolved by applying filters for location. We suspect the cause of this is the length of the documents of Wikipedia articles in comparison to the results from the travel forums. The length of a document being favoured is a general trend we observe using term-based search. Using a ranking method such as BM25, using a semantic based approach to searching could be a method to resolve these issues. Finally a more balanced approach would have been to scrape multiple-location across domains instead of matching a single continent to a scraped domain.

2.5 Refreshing the repository

During real life deployment it would be a good idea to be able to regularly update the repository. This can simply be achieved by rerunning the crawling procedure as described in the previous section, and then deal with one of the two options: 1) the url is already in the *url_list_large.txt* file, but the HTML has been updated or 2) the url is not yet in the *url_list_large.txt* file.

We first tried to implement a system to deal with scenario 1, by first checking if the text from the HTML body is different from the old HTML document and then update if the HTML is different. While a legitimate strategy, this results in practically checking the majority of the pages in the old repository, resulting in a very slow refresh. It was therefore simply faster to just replace the old HTML text with the 'new' HTML text, even if they were actually still the same.

For option 2, when a URL is not yet in *url_list_large.txt*, it can simply added to the file (at the same time a inverted index is created for this new page content and added to the larger index, this will be further described in the following section).

3 DATA PROCESSING AND STORAGE

Once the chosen domains have been crawled and the contents of the visited pages have been scraped, we further process the data to allow users to search the scraped pages. This section covers the processing of the data and indexing that processed data. For data processing and indexing of the crawled pages we have thus far implemented three basic functionalities: firstly, this is the preprocessing pipeline, secondly the single-thread indexing and thirdly distributed indexing. Each functionality and its implementation shall be described briefly in the following sections. As stated in the introduction, all the code used can be found on the public GitHub repository. More specifically, the code for this section can be found in the *Indexing_preprocessing.py* file in the *Indexing* folder.

3.1 Preprocessing

The first step in the preprocessing pipeline involves in extracting all the text from the gathered HTML pages. Here we delete all script and style elements from the HTML, remove all the characters like punctuation and newlines. This results in a string of text from each page. These strings are then further processed with the help of the NLTK Python module, which is a library for working with text[5]. First the strings are made lowercase, then they get tokenized - which means splitting documents into vectors of individual words - with the NLTK tokenizer function, then the stop words are removed which belong the the built-in English stop word list from NLTK. After this all tokens are stemmed with the Porter stemmer. The result is a list with the cleaned tokens in the right order for every page.

[5]https://www.nltk.org/

3.2 Single index for all information

Basic search functionality is underpinned by the concept of taking words from the query and looking up documents that contain these words. To allow for this functionality, an inverted index of all documents in the corpus has to be created where each word in the vocabulary of the corpus resides together with pointers to the documents in which they appear, as well as other information like the location of the given word or the total frequency. This step is necessary in order to perform queries, retrieve documents and ranking results of the user query. As mentioned in the previous section, each document is now a list of tokens, and we first want to be able to index these on a single machine. This is done with the help of multiple self created functions which can be found in the *Indexing.py* and *preprocessing.py* file. Indexing with self made functions has the benefit of having full control of the data structures, file output and the information stored in the index. However, this approach also has some caveats, the main being that searching in this index is not as fast as conventional search engines usually are, and that the index files have to be loaded upon making a query in the interface. A faster approach is available with solutions such as the popular search and analytics framework Elasticsearch.

An index is created with the structure displayed in figure 1:

```
{
'word1': {'doc1':[n_appearances, [index1, index2]],
    'doc3': [n_appearances, [index1, index2]]},
'word2': {'doc2':[n_appearances, [index1, index2, index3]]}
}
```

Figure 1: Dictionary structure of indexed webpages.

The document ID is the URL of the given page, this ensures that each document ID is unique. At this moment the frequencies with the positions per document is stored, this could later easily be supplemented with scores like TF-IDF.

3.3 Distributed indexing

As the collection of data would grow, it would be important to have the functionality of distributed indexing to speed up this process. This can be done by partitioning the list of URLs that were scraped by the spiders. The entire process as described 3.1 can then be performed on different CPUs. Because the index is the URL of the given page, and the file containing URLs was partitioned, it is guaranteed that each ID is unique during the distributed indexing. The resulting indices can be merged on a single machine with a function which can be found in the aforementioned Python file. The result is one single index.

3.4 Indexing title and body separately

In addition to the indexing of the information from the body of the pages, the title of the pages were indexed separately. Using the BeautifulSoup package, the title and body were extracted consecutively from the URLs. Thereafter, two separate indices were created derived from the extracted titles and bodies. Please refer to the folder 'Indexing' on the Github page, where the related code can be found in the following file: *seperate_indexes_title_and_body.py*

3.5 Extraction of additional information

Several functionality can be added to the design of our search engine to make finding information more convenient. Additional information was extracted indicating whether or not a certain page is about one of the following topics: food, culture or transportation. Moreover, combination of these topics are possible. To determine whether a page belongs to a topic, vocabularies related to food, culture and transportation were derived from [6] and [7]. These vocabularies were converted into three separate topic lists, containing dozens of words for each topic. Moreover, the terms in every list were processed and stemmed in accordance with the query and document processing and stemming. Whenever a query is entered to the search engine and a radio button is selected, first the URLs of documents that contain more than 100 unique words (this threshold can be changes according to preference) from the related topic list are added to a list. Subsequently, only the documents in that list are considered and ranked with the vector-space model. This results in a ranked list of the documents related to the desired topic, which could be either food, culture or transportation (or a combination thereof).

3.6 Updating the index

As with updating the repository, which was explained in section 2.4, the index must also be updated accordingly. Here too, there are two scenarios we can encounter: 1) a new page which was not yet in the existing inverted index is added, or 2) a page is updated and the information for that index needs to be updated.

For scenario 1, the inverted index can be loaded into local memory, then create a new index for the given text from the new URL, and use the merge function in the *Indexing_preproccessing.py* in the indexing folder on the Github repository.

For scenario 2, the same steps as for scenario 1 are repeated, with the extra step of deleting the information of the old HTML text from the inverted index. So first the inverted index is loaded into memory, then all the old information for that given URL is deleted from that index, then a new index is created for the new/updated page is created and merged the same way as in scenario 1.

4 BASIC SEARCH

For the first basic implementation of search we have implemented a system which processes a query the same way as the documents covered in the previous sections and then ranks the documents based on the cosine similarity of the TF-IDF vector representations of the query and the documents.

[6]http://www.english-for-students.com/Food-and-Eating-Vocabulary.html
[7]https://relatedwords.org/relatedto/culture

4.1 Query processing

As the query is to be compared to documents in the form of term vectors, the preprocessing has to be identical to the documents. The search Python file on the public GitHub contains the function which takes a string as input, e.g. "Holiday Vietnam!". The query gets processed, spell checked and then transformed to a vector with TF-IDF values for each word:

```
1) query string as input: 'Holidaay Vietnam!'
2) tokenized and cleaned: ['holidaay', 'vietnam']
3) spelling correction: ['holiday', 'vietnam']
3) Vector with with TF-IDF values:  (0.001, 0.0034)
```

The first step performs the same preprocessing steps as the documents in the previous sections: It is made lowercase, remove special characters, tokenize, and then perform stemming. In addition, the query gets spell checked. The spell check occurs after tokenization and before stemming. The result is a list with the spell-checked, stemmed and cleaned words. The tokens then need to be transformed to the TF-IDF vector. The TF-IDF values can be can be calculated with the metrics in that are stored in the inverted index. Then, for all the URLs in the inverted index which contain at least one word from the query, a TF-IDF vector is created (with a value of zero when the word, e.g. Vietnam, does not appear in the document). These vector representations can then be used to rank based on the cosine circularity. T

4.2 Term-based Search

As was described before we implemented a vector space model for retrieving and ranking the documents given a query. In the vector space model, each document which contains at least one word from the query is represented as a word vector. Each vector has the size of the query, i.e. number of words in the query. We also tried to use the entire vocabulary of the corpus as the vector size, but as the vocabulary size was larger than 200.000, the calculations took too long and loading the all the vector representation into memory was also not possible. This is why we decided to take the query vocabulary as the vector representation. When a given word is present in document, the value is 1 or a metric like TF-IDF (as is implemented here), or a 0 when the word is not present. The vector representation of documents allows for similarity matching by calculating the cosine similarity for each document and a given query (as described in the previous section). The documents can then simply be ordered and returned by this metric. We choose to implement TF-IDF values, as these are more informative then simple binary of TF word metrics. The TF-IDF values where calculated with the use of and inserted in the inverted index. The search function can be found in the *search.py* file in the *flaskr* folder on the GitHub repository.

4.3 Semantic Search

Moreover, we implemented semantic search as search algorithm for Travelsearch. To do so, we enhanced the aforementioned term-based search method with semantic-based techniques. The method that was implemented utilizes Word Embedddings to find semantically similar words for a given query. This can be done by using a trained Word Embeddings model to find words for a given query word that are closest in the embedding space. So for example for the word 'country', the top three words that are closest in the Word Embedding space are 'nation', 'continent' and 'region'. These three words would be relevant if they are present in a document for someone who is searching for a 'country' related topic.

The pretrained word2vec model that was used was trained on 100 billion words from a Google News dataset and the vocabulary of the model contains about 3 million words and phrases [8]. The assumption is made that the substantial majority of the vocabulary of the corpus of Travelsearch will be in this 3 million words sized vocabulary of the word2vec model. Adding words to the query that are semantically similar will hopefully result in documents that contain more content on the topic mentioned in the query will be ranked higher.

The steps in the query processing and the vector space model remain the same, the only difference is that the additional words as determined by the similarity in the Word Embedding space are added to the query vector. So first a query is entered into the system, then for each (non-stemmed) word in the query, the top 3 simular words in the Word Embedding space are added to the query, and then the ranking is performed as described in the previous sections. After experimenting with different settings, the top three of the most semantically similar words seem to work best.

The Semantic Search method works as follows:

```
1) query string as input: 'Hello World'
2) tokenized and cleaned: ['hello', 'world']
3) Add top 3 similar words to for each query word:
      ['hello', 'hi', 'goodby', 'howdi',
           'world', 'globe', 'countri', 'theworld']
3) Vector with with TF-IDF values:
      (0.001, 0.0034, 0.029,
              0.0034, 0.012, 0.0034)
```

However, in case of Travelsearch implementing semantic search did yield significant superior results compared to the simpler term-based method. Therefore, there was chosen to stick with the term-based search as the final search algorithm, because this function was faster. The implementation of Semantic Search can be found in the semantic_search.py file in the *flaskr* folder on the GitHub repository. Specifically, the code for the semantic methods starts on line 206.

[8] https://code.google.com/archive/p/word2vec/

5 ADDITIONAL SEARCH FEATURES AND INTERFACE

5.1 Interface

The interface for the search engine was built with Flask[9] to facilitate straightforward communication between the ranking scripts and the front-end. The basic Flask functionality works by defining routing and template options within a Python script that runs the Flask application. Within this script, Python variables can be passed to different html templates to render webpages.

The *base.html* template contains all necessary html code that is the same for every subdirectory (JavaScript and CSS dependencies, navigation bar, header). The default home page shown in 2 is loosely based on traditional search engine interfaces such as Google, Bing or Yandex. The interface is made with Bootstrap 4, which provides static CSS resources and templates. The theme used for this project is free and available for download on Bootswatch[10]. When a user enters or clicks the 'search' button, the query in the text field requests the ranked documents from the application Python script[11].

The results of the out-of-the-box search as described in section 4.2 are stored in a list of dictionaries which is then passed to the Flask function *render_template*(). The dictionaries contain page title, url and the body text of the page. By passing variables to this function, these variables can be used in the html templates to display the results of the query. To limit the amount of information displayed on the screen, we choose to display anywhere between 5 and 20 results. An optimum amount of results to display on the screen were chosen after more refined ranking algorithms are implemented. To match the amount of results with the evaluation part of the project, we settled on displaying ten results per page. Within the time frame of this project, we did not have enough time to implement a pagination feature to load more results.

On the results page as shown in 3, we display the most relevant results first without any filtering applied. The top line shows the title of the web page, followed by the URL and page text in which the query word is mentioned (query highlighting). The radio buttons each send POST requests in the same way the results were loaded when searching from the home page.

One of the challenges and eventual shortcomings of the chosen approach using Flask is the lack of a database structure that is queried upon sending a request. The consequence of this is that every time the user sends a query, the index file has to be loaded in the script that runs the application. Loading and searching the index file in our implementation would take too long to be a usable product. Faster query speeds are possible using systems such as Elastic Search. This could also be resolved by hosting the index files on a server or using a database solution. The latter option is often used in Flask applications.

[9]http://flask.pocoo.org/docs/1.0/

[10]https://bootswatch.com/

[11]https://github.com/jessefh/travelsearch/blob/master/flaskr/travelsearch.py

5.2 Additional features

To facilitate convenient search for users who are only interested in documents about a specific topic, we created radio buttons that filter for the indicated topic(s) as shown in 4. Any combination of these three radio buttons can be selected by a user if the user is interested in the associated topic. The search engine will then only return documents about the selected topic by looking in the document for a list of certain keywords related to that topic as mentioned in section 3.5. We implemented this 'checkbox' feature for the topics food, culture and transportation, as these might be relevant classes when planning for or during a holiday. As previously discussed, a document is assumed to be about a certain topic, when it contains more words from the associated topic list than a specified threshold (for instance more than 100 distinct words). The code for these features can be found on the GitHub repository in the search.py file in the folder 'Search'.

In addition, a feature was implemented which helps users narrow down the search results to their desired continent: either Asia, Europe or South America, from which only one can be chosen. The interface for this feature is shown in figure 5 To do so, prior knowledge of the characteristics of the domains was utilized, since we know that certain domains are only associated with certain continents. The crawled domains Fodors, RickSteves and Lonely Planet are respectively associated with Asia, Europe and South America. Therefore, when a user specifies one of the continents, the search will only be performed on the pages that are associated with the relevant domain. The code for this feature also can be found on the GitHub repository in the *search.py* file in the folder 'Search'.

Moreover, an evaluation feature was created as shown in 6 and integrated with the interfaced. Besides every page in the ranking there are check boxes shown, in which users can indicate whether the associated page is relevant or irrelevant. Upon clicking the submit button, a line will be written in a text file. Documents from 1 to 10 will then be labeled 'R' if they are relevant, and 'N' if not relevant. The results of these evaluation forms are discussed in further detail in section 6.

6 EVALUATION

6.1 Gathering Relevance Judgements

Due to time constraints we did not use the additional evaluation feature of our search engine, instead a Google Forms[12] was created as an interface to gather relevance judgements. The Google Forms provides instructions for the assessors on to the nature of the queries and how to assess them. Moreover, all 20 queries and the resulting 10 highest ranked documents are listed in the Google Forms, in which the assessors can indicate whether a document is relevant or not. Eventually three groups have assessed our queries: group 10, group 16 and group 19. Finally, we helped to other teams by providing relevance judgements for their queries.

[12]https://docs.google.com/forms/d/17ptN7vN-nd7NEZFvFM-DUP0JFEEGIW80TqYwXKly4sk/edit?ts=5c9bbd46

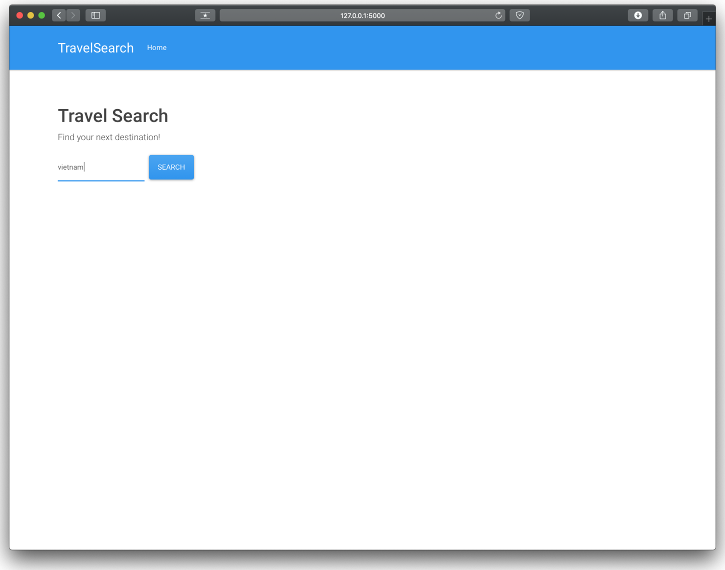
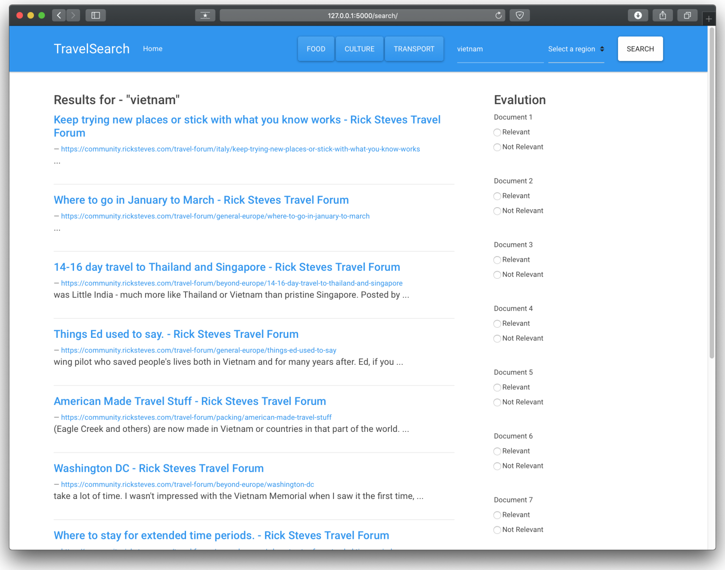


Figure 2: Screenshot of the basic search engine landing page.



Results for - "vietnam"

Keep trying new places or stick with what you know works - Rick Steves Travel Forum
— https://community.ricksteves.com/travel-forum/italy/keep-trying-new-places-or-stick-with-what-you-know-works
...

Where to go in January to March - Rick Steves Travel Forum
— https://community.ricksteves.com/travel-forum/general-europe/where-to-go-in-january-to-march
...

14-16 day travel to Thailand and Singapore - Rick Steves Travel Forum
— https://community.ricksteves.com/travel-forum/beyond-europe/14-16-day-travel-to-thailand-and-singapore
was Little India - much more like Thailand or Vietnam than pristine Singapore. Posted by ...

Things Ed used to say. - Rick Steves Travel Forum
— https://community.ricksteves.com/travel-forum/general-europe/things-ed-used-to-say
wing pilot who saved people's lives both in Vietnam and for many years after. Ed, if you ...

American Made Travel Stuff - Rick Steves Travel Forum
— https://community.ricksteves.com/travel-forum/packing/american-made-travel-stuff
(Eagle Creek and others) are now made in Vietnam or countries in that part of the world. ...

Washington DC - Rick Steves Travel Forum
— https://community.ricksteves.com/travel-forum/beyond-europe/washington-dc
take a lot of time. I wasn't impressed with the Vietnam Memorial when I saw it the first time, ...

Where to stay for extended time periods. - Rick Steves Travel Forum

Figure 3: Screenshot of results page.

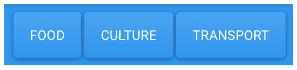


Figure 4: Radio buttons that filter results.

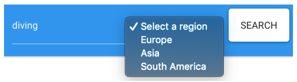
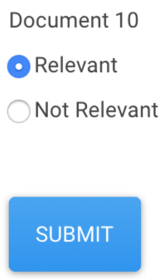


Figure 5: Drop down menu showing possible location filters.


Document 10

[✓ Relevant] Not Relevant

Figure 6: Example evaluation radio button of tenth document with submit.

6.2 Offline Metrics

The search engine was evaluated based on the following four offline metrics: Precision, Reciprocal Rank (RR), P@3 and P@5. The results are displayed in Table 2. Evaluation 1, 2 and 3 were done by groups 19, 10 and 16, respectively. The column "Options" displays which options were turned on at that specific query. "F" means food was turned on, "T" means transportation was turned on, and C means culture was turned on. Furthermore "C=E" means continent = 'Europe' was turned on. The data that was used to calculate these metrics can be found in the *relevance_judgements.csv* file in the 'Evaluation' folder in the GitHub repository. As can be seen from Table 2, the quality of search results varies a lot between queries, i.e. the degree to which the returned documents satisfy the user's query intent highly varies for different queries. In general, the relevance of the returned documents is rather low. The reason for this is that the documents, forum pages, are really long and therefore contain a great many words. Since we are using term-based search, it is possible that a document contains the query term many times even though it is not the main topic of the document. In addition, since the documents are really long, assessors arguably did not read the whole document. This could result in assessors assessing documents as non-relevant, even though they actually contain relevant information in terms of the query intent. Moreover, Table 2 shows that for most queries precision3 and precision5 are higher than overall precision (precision10). This implies that the highest ranked documents, respectively top 3 and top 5, have a higher quality than all 10 documents returned. It is more important that higher ranked documents are relevant than the lower ranked documents, so this is a good feature of the search engine. Furthermore, the Reciprocal Rank (RR) for most queries is 1, which implies that the highest ranked document by Travelsearch in most cases is indeed a relevant document.

6.3 Inter-Assessor Agreement

To determine the inter-assessor agreement the pairwise Cohen's Kappa coefficients were computed for all three assessor pairs. Since there were more than two assessors, the average of the pair-wise coefficients was calculated as well. The results can be found in Table 1.

	Cohen's Kappa coefficient
Group 19 & 10	0.45
Group 10 & 16	0.38
Group 19 & 16	0.29
Average	0.37

Table 1: "Cohen's Kappa coefficient"

Table 1 shows that inter-assessor agreement of the three assessors is low (<0.67). Since the degree to which assessors agree about relevance judgements is low, not to many conclusions should be drawn from the current offline metrics. To address this problem of low inter-assessor agreement, we would recommend to gather more relevance judgements of other assessors in the future.

			Precision			RR			P@3			P@5		
N	Query String	Options	Eval1	Eval2	Eval3	Eval1	Eval2	Eval3	Eval1	Eval2	Eval3	Eval1	Eval2	Eval3
1	"Planning trip to Argentina"		0.8	0.7	0.8	1	0.5	1	1	0.66	1	0.8	0.6	0.6
2	"Rome"	F	0.2	0.3	0.3	0.5	0.5	1	0.33	0.33	0.66	0.2	0.2	0.4
3	"Vietnam Hanoi to Hue"	T	0.6	0.9	0.5	1	1	1	1	1	1	1	1	0.8
4	"Oktoberfest "	C=E	0.7	0.7	0.2	1	1	0.2	0.66	0.66	0	0.8	0.8	0.2
5	"Boat trip"	T, C=E	0.5	0.2	0.1	1	1	1	0.33	0.33	0.33	0.4	0.4	0.2
6	"art"	C, C=E	0.3	0.3	0.3	0.33	0.33	0.33	0.33	0.33	0.33	0.4	0.4	0.4
7	"Japan"		0.3	0.1	0.1	0.25	0.25	0.25	0	0	0	0.2	0.2	0.2
8	"Oslo"		0.3	0.5	0.3	1	1	1	0.33	0.66	0.66	0.2	0.6	0.4
9	"Tokyo"		0.2	0.2	0.3	0.5	0.5	0.5	0.33	0.33	0.33	0.2	0.2	0.2
10	"Athens"		0.4	0.5	0.5	1	1	1	1	1	1	0.6	0.6	0.8
11	"Kreta"		0.5	0	0.1	1	0	0.2	1	0	0	0.6	0	0.2
12	"London"		0.5	0.5	0.3	1	1	0.33	0.66	0.66	0.33	0.2	0.6	0.4
13	"Hiking"		0.5	0.5	0.1	1	1	1	0.66	0.66	0.33	0.6	0.6	0.2
14	"Restaurants"		0.3	0.5	0.3	0.33	1	0.5	0.33	0.33	0.66	0.4	0.6	0.4
15	"Biking"		0.4	0.2	0	0.5	0.5	0	0.33	0.33	0	0.2	0.2	0
16	"Sushi"		0.4	0.4	0.1	1	1	1	1	0.66	0.33	0.6	0.4	0.2
17	"Nightlife"		0.3	0.6	0	0.14	0.5	0	0	0.5	0	0	0.6	0
18	"Beaches"		0.6	0.5	0.1	1	1	0	0.66	0.66	0	0.6	0.6	0
19	"Eating out"		0.2	0.5	0.2	0.25	0.33	0.25	0	0.33	0	0.2	0.4	0.2
20	"Art"	C	0.3	0.3	0.2	0.33	0.33	0.33	0.33	0.33	0.33	0.6	0.4	0.4

Table 2: "Four offline evaluation metrics"